

RESTful Web Services in Java.

目錄

用Jersey构建RESTful服务

> 作者：waylau
>
> 来源：RestDemo

用Jersey构建RESTful服务1--HelloWorld

一、环境

1. Eclipse Juno R2
2. Tomcat 7
3. Jersey 2.x(最新2.11版本测试通过) 下载地址（https://jersey.java.net/download.html）

二、流程

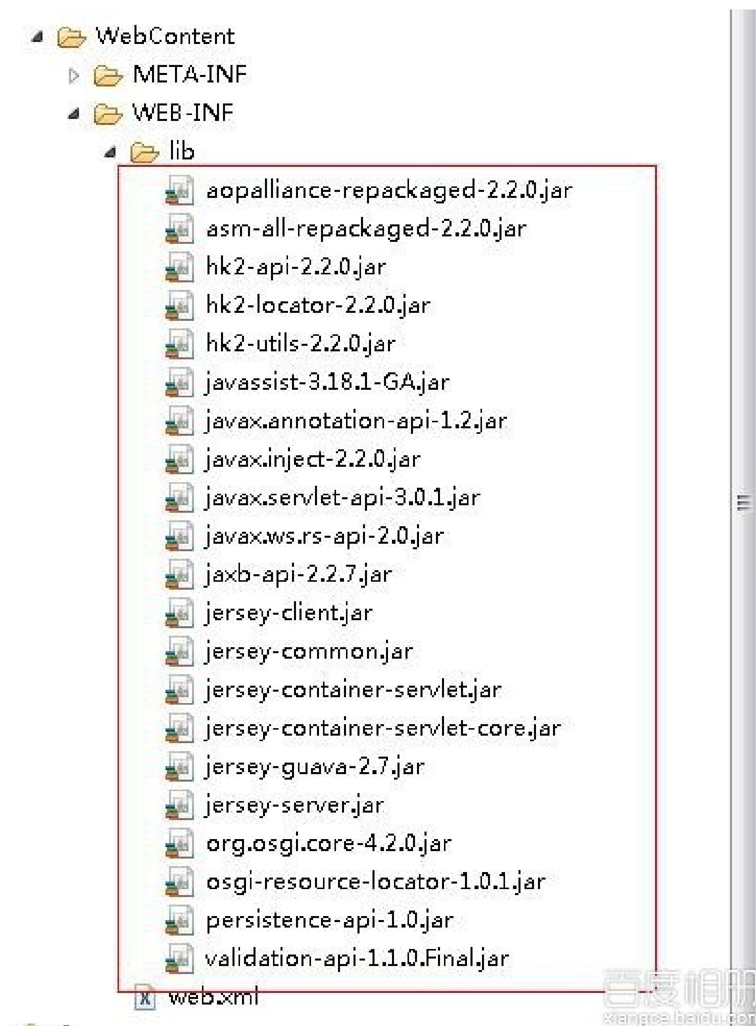
1. Eclipse 中创建一个 Dynamic Web Project ,本例为“RestDemo”
2. 按个各人习惯建好包，本例为“com.waylau.rest.resources”



3. 解压jaxrs-ri-2.7，将api、ext、lib文件夹下的jar包都放到项目的lib下；



项目引入jar包



4. 在resources包下建一个class“HelloResource”

```
package com.waylau.rest.resources;
   import javax.ws.rs.GET;
   import javax.ws.rs.Path;
   import javax.ws.rs.Produces;
   import javax.ws.rs.PathParam;
   import javax.ws.rs.core.MediaType;
   @Path("/hello")
   public class HelloResource {
       @GET @Produces(MediaType.TEXT_PLAIN)
       public String sayHello() {
           return "Hello World!" ;
       }

       @GET @Path("/{param}")
       @Produces("text/plain;charset=UTF-8")
       public String sayHelloToUTF8(@PathParam("param") String
           return "Hello " + username;
       }

   }
```

5. 修改web.xml,添加基于Servlet-的部署

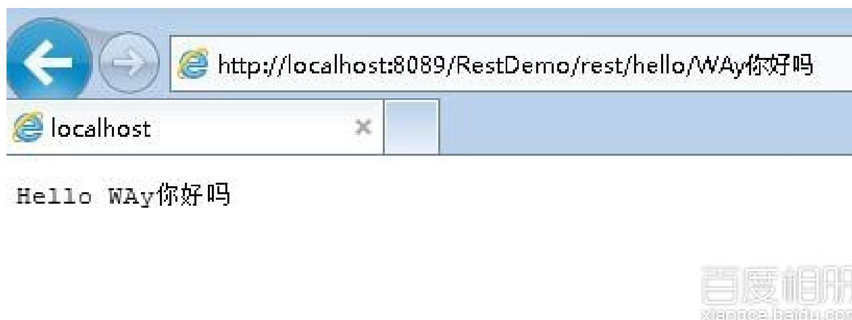
```
<servlet>
    <servlet-name>Way REST Service</servlet-name> <servlet-clas
    <init-param>
    <param-name>jersey.config.server.provider.packages</param-r
    <param-value>com.waylau.rest.resources</param-value> </init
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>Way REST Service</servlet-name>
    <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

6. 项目部署到tomcat,运行
7. 浏览器输入要访问的uri地址 http://localhost:8089/RestDemo/rest/hello,输出 Hello World!



http://localhost:8089/RestDemo/rest/hello/Way%E4%BD%A0%E5%A5%BD%E5%90%97,输出Hello Way你好吗



参考：https://jersey.java.net/documentation/latest/user-guide.html

本章源码：https://github.com/waylau/RestDemo/tree/master/jersey-demo1-helloworld

用Jersey构建RESTful服务2--JAVA对象转成XML输出

一、 总体说明

XML和JSON 是最为常用的数据交换格式。本例子演示如何将java对象，转成XML输出。

二、流程

1. 在上文的例子中，创建一个包“com.waylau.rest.bean”
2. 在该包下创建一个JAVA类"User"

```
package com.waylau.rest.bean;
    import javax.xml.bind.annotation.XmlRootElement;
    /* *
     * 用户 bean
     * @author waylau.com
     * 2014-3-17
     */
    @XmlRootElement
    public class User {

        private String userId;
        private String userName;
        private String age;

        public User() {};

        public User(String userId, String userName, String age)
            this.userId = userId;
            this.userName = userName;
            this.age = age;
        }
        public String getUserId() {
            return userId;
        }
        public void setUserId(String userId) {
            this.userId = userId;
        }
        public String getUserName() {
            return userName;
        }
        public void setUserName(String userName) {
            this.userName = userName;
        }
        public String getAge() {
            return age;
        }
        public void setAge(String age) {
            this.age = age;
        }
    }
```

注意：该类上面增加了一个注解“@XmlRootElement”，在将该类转化成XML时，说明这个是XML的根节点。

3. 在“com.waylau.rest.resources”中，增加资源“UserResource“，代码如下：

```
package com.waylau.rest.resources;

    import java.util.ArrayList;
    import java.util.HashMap;
```

```
    import java.util.List;
    import java.util.Map;

    import javax.ws.rs.Path;
    import javax.ws.rs.Produces;
    import javax.ws.rs.PathParam;
    import javax.ws.rs.core.MediaType;
    import javax.ws.rs.DELETE;
    import javax.ws.rs.GET;
    import javax.ws.rs.POST;
    import javax.ws.rs.PUT;

    import com.waylau.rest.bean.User;

    @Path("/users")
    public class UserResource {
        private static Map<String,User> userMap = new HashMap<S
         /**
         * 查询所有
         * @return
         */
        @GET
        @Produces(MediaType.APPLICATION_XML)
        public List<User> getAllUsers(){
            List<User> users = new ArrayList<User>();
            User u1 = new User("001","WayLau","26");
            User u2 = new User("002","King","23");
            User u3 = new User("003","Susan","21");

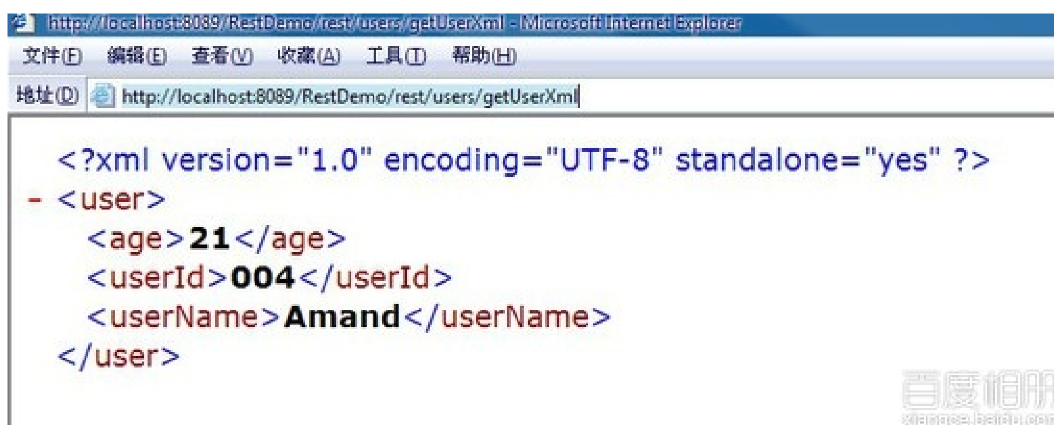
            userMap.put(u1.getUserId(), u1);
            userMap.put(u2.getUserId(), u2);
            userMap.put(u3.getUserId(), u3);

            users.addAll( userMap.values() );
            return users;
        }

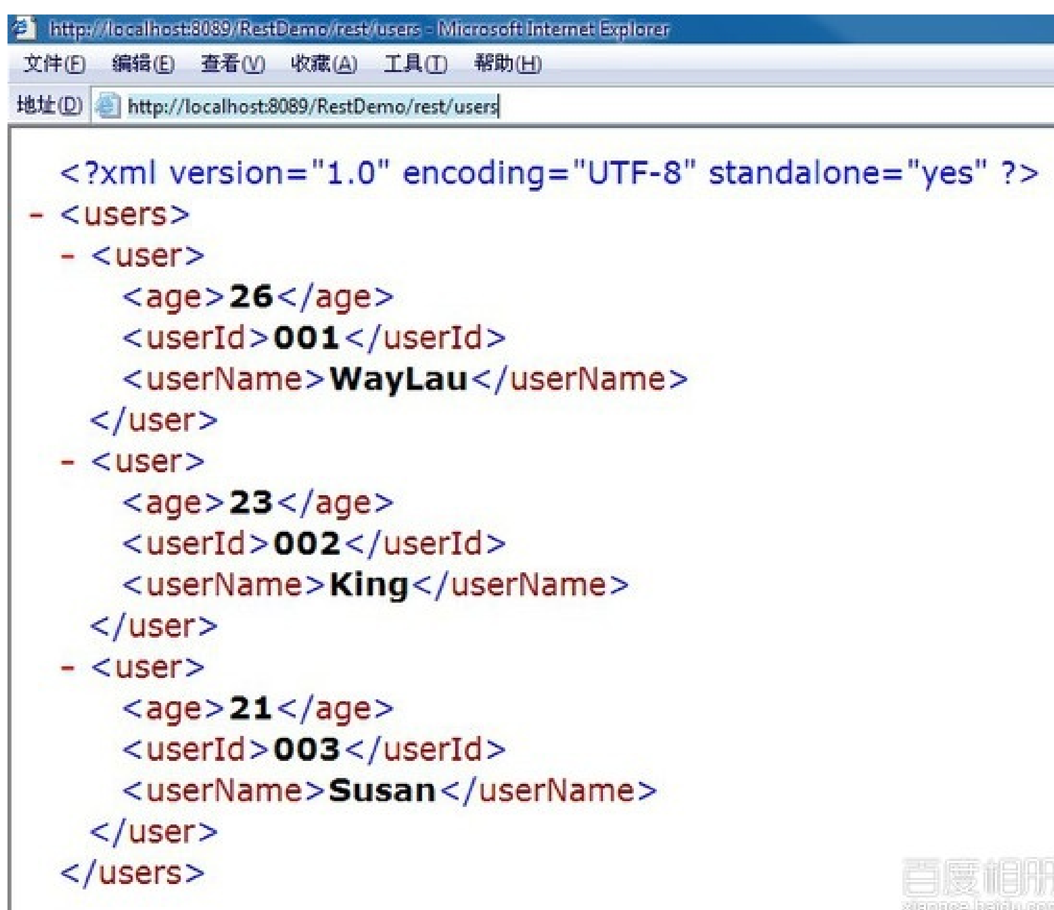
        @GET
        @Path("/getUserXml")
        @Produces(MediaType.APPLICATION_XML)
        public User getUserXml() {
         User user  = new User();
         user.setAge("21");
         user.setUserId("004");
         user.setUserName("Amand");
         return user;
        }
    }
```

其中MediaType.APPLICATION_XML 说明了是以XML形式输出

在浏览器输入http://localhost:8089/RestDemo/rest/users/getUserXml，输出单个对象



在浏览器输入 http://localhost:8089/RestDemo/rest/users 输出对象的集合



本章源码：https://github.com/waylau/RestDemo/tree/master/jersey-demo2-xml

用Jersey构建RESTful服务3--JAVA对象转成JSON输出

一、 总体说明

XML和JSON 是最为常用的数据交换格式。本例子演示如何将java对象，转成JSON输出。

二、流程

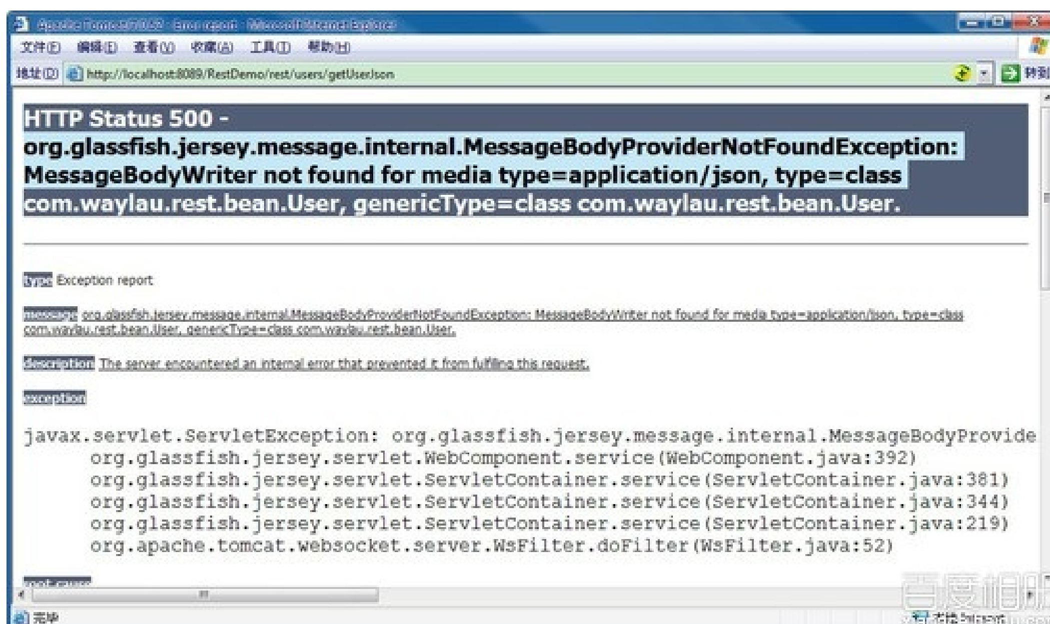
1. 在上文项目中， 在“com.waylau.rest.resources.UserResource“中增加代码，代码如下：

   ```
   @GET
      @Path("/getUserJson")
      @Produces(MediaType.APPLICATION_JSON)
      public User getUserJson() {
       User user  = new User();
       user.setAge("27");
       user.setUserId("005");
       user.setUserName("Fmand");
       return user;
      }
   ```

   MediaType.APPLICATION_JSON 说明输出的是JSON格式

2. 运行项目，浏览器输入http://localhost:8089/RestDemo/rest/users/getUserJson 期望获取到json的数据，此时，项目报错

```
org.glassfish.jersey.message.internal.MessageBodyProviderNotFc
        at org.glassfish.jersey.message.internal.WriterIntercep
        at org.glassfish.jersey.message.internal.WriterIntercep
        at org.glassfish.jersey.filter.LoggingFilter.aroundWrit
        at org.glassfish.jersey.message.internal.WriterIntercep
        at org.glassfish.jersey.server.internal.JsonWithPadding
        at org.glassfish.jersey.message.internal.WriterIntercep
        at org.glassfish.jersey.server.internal.MappableExcepti
        at org.glassfish.jersey.message.internal.WriterIntercep
        at org.glassfish.jersey.message.internal.MessageBodyFac
        at org.glassfish.jersey.server.ServerRuntime$Responder.
        at org.glassfish.jersey.server.ServerRuntime$Responder.
        at org.glassfish.jersey.server.ServerRuntime$Responder.
        at org.glassfish.jersey.server.ServerRuntime$1.run(Serv
```



此时，需要获取json转换包的支持 。 可以由多种方式实现：MOXy、JSON-P、Jackson、Jettison等，本例为Jackson。

3. jackson-all-1.9.11.jar 下载地址http://wiki.fasterxml.com/JacksonDownload
4. 项目中引入jackson-all-1.9.11.jar
5. 在“com.waylau.rest”目录下创建RestApplication.java

```
package com.waylau.rest;

import org.codehaus.jackson.jaxrs.JacksonJsonProvider;
import org.glassfish.jersey.filter.LoggingFilter;
import org.glassfish.jersey.server.ResourceConfig;

/**
 * 应用
 * @author waylau.com
 * 2014-3-18
 */
public class RestApplication extends ResourceConfig {
    public RestApplication() {

     //服务类所在的包路径
     packages("com.waylau.rest.resources");
     //注册JSON转换器
     register(JacksonJsonProvider.class);

    }
}
```

6. 修改web.xml，初始化从RestApplicaton进入应用，如下：

```
<servlet>
     <servlet-name>Way REST Service</servlet-name>
     <servlet-class>org.glassfish.jersey.servlet.ServletContair
      <init-param>
           <param-name>javax.ws.rs.Application</param-name>
           <param-value>com.waylau.rest.RestApplication</param-
       </init-param>

    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>Way REST Service</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
```

7. 运行项目，再次访问http://localhost:8089/RestDemo/rest/users/getUserJson 即可输出JSON文本



本章源码：https://github.com/waylau/RestDemo/tree/master/jersey-demo3-json

用Jersey构建RESTful服务4--通过jersey-client客户端调用Jersey的Web服务模拟CURD

一、总体说明

通过jersey-client接口，创建客户端程序，来调用Jersey实现的RESTful服务，实现增、删、改、查等操作。 服务端主要是通过内存的方式，来模拟用户的增加、删除、修改、查询等操作。

二、创建服务端

1. 在上文项目中， 在“com.waylau.rest.resources.UserResource“中修改代码，首先创建一个HashMap，用来保存添加的用户

```
private static Map<String,User> userMap  = new HashMap<String,U
```

2. 创建增、删、改、查 用户资源等操作

```
/**
     * 增加
     * @param user
     */
    @POST
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION
    public void createStudent(User user) {
        userMap.put(user.getUserId(), user );
    }

    /**
     * 删除
     * @param id
     */
    @DELETE
    @Path("{id}")
    public void deleteStudent(@PathParam("id")String id){
        userMap.remove(id);
    }

    /**
     * 修改
     * @param user
     */
    @PUT
```

```
    @Consumes(MediaType.APPLICATION_XML)
    public void updateStudent(User user){
        userMap.put(user.getUserId(), user );
    }

    /**
     * 根据id查询
     * @param id
     * @return
     */
    @GET
    @Path("{id}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION
    public User getUserById(@PathParam("id") String id){
        User u = userMap.get(id);
        return u;
    }

    /**
     * 查询所有
     * @return
     */
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION
    public List<User> getAllUsers(){
        List<User> users = new ArrayList<User>();
        users.addAll( userMap.values() );
        return users;
    }
```

三、创建客户端程序

创建包“com.waylau.rest.client”，在包下建一个UserClient.java，代码如下：

```
package com.waylau.rest.client;

    import javax.ws.rs.client.Client;
    import javax.ws.rs.client.ClientBuilder;
    import javax.ws.rs.client.Entity;
    import javax.ws.rs.client.WebTarget;
    import javax.ws.rs.core.MediaType;
    import javax.ws.rs.core.Response;

    import org.codehaus.jackson.jaxrs.JacksonJsonProvider;

    import com.waylau.rest.bean.User;

    /**
     * 用户客户端，用来测试资源
```

```
     * @author waylau.com
     * 2014-3-18
     */
    public class UserClient {

        private static String serverUri = "http://localhost:8089/Re
        /**
         * @param args
         */
        public static void main(String[] args) {
            addUser();
            getAllUsers();
            updateUser();
            getUserById();
            getAllUsers();
            delUser();
            getAllUsers();

        }
        /**
         * 添加用户
         */
         private static void addUser() {
             System.out.println("****增加用户addUser****");
             User user = new User("006","Susan","21");
             Client client = ClientBuilder.newClient();
             WebTarget target = client.target(serverUri + "/users")
             Response response = target.request().buildPost(Entity.
             response.close();
        }

        /**
         * 删除用户
         */
         private static void delUser() {
             System.out.println("****删除用户****");
             Client client = ClientBuilder.newClient();
             WebTarget target = client.target(serverUri + "/users/0
             Response response = target.request().delete();
             response.close();
        }

        /**
         * 修改用户
         */
         private static void updateUser() {
             System.out.println("****修改用户updateUser****");
             User user = new User("006","Susan","33");
             Client client = ClientBuilder.newClient();
             WebTarget target = client.target(serverUri + "/users")
             Response response = target.request().buildPut( Entity.
             response.close();
        }
```

```
        /**
         * 根据id查询用户
         */
         private static void getUserById() {
             System.out.println("****根据id查询用户****");
             Client client = ClientBuilder.newClient().register(Jac
             WebTarget target = client.target(serverUri + "/users/0
             Response response = target.request().get();
             User user = response.readEntity(User.class);
             System.out.println(user.getUserId() + user.getUserName
             response.close();
        }
        /**
         * 查询所有用户
         */
         private static void getAllUsers() {
             System.out.println("****查询所有getAllUsers****");

             Client client = ClientBuilder.newClient();

             WebTarget target = client.target(serverUri + "/users")
             Response response = target.request().get();
             String value = response.readEntity(String.class);
             System.out.println(value);
             response.close();  //关闭连接
         }

    }
```

四、运行

启动服务端项目，运行客户端程序UserClient，控制台输出如下

```
****增加用户addUser****
****查询所有getAllUsers****
[{"userId":"006","userName":"Susan","age":"21"}]
****修改用户updateUser****
****根据id查询用户****
006Susan33
****查询所有getAllUsers****
[{"userId":"006","userName":"Susan","age":"33"}]
****删除用户****
****查询所有getAllUsers****
[]
```

五、总结

1. 客户端如果需要进行JSON转换，需要进行JSON注册

```
Client client = ClientBuilder.newClient().register(JacksonJsonP
```

2. WebTarget 指明了要请求的资源的地址
3. target.request(). 后面跟的是请求的方法:POST，GET，PUT或DELETE

本章源码：https://github.com/waylau/RestDemo/tree/master/jersey-demo4-client-curd

用Jersey构建RESTful服务5--Jersey+MySQL5.6+Hibernate4.3

一、总体说明

本例运行演示了用Jersey构建RESTful服务中，如何同过Hibernate将数据持久化进MySQL的过程

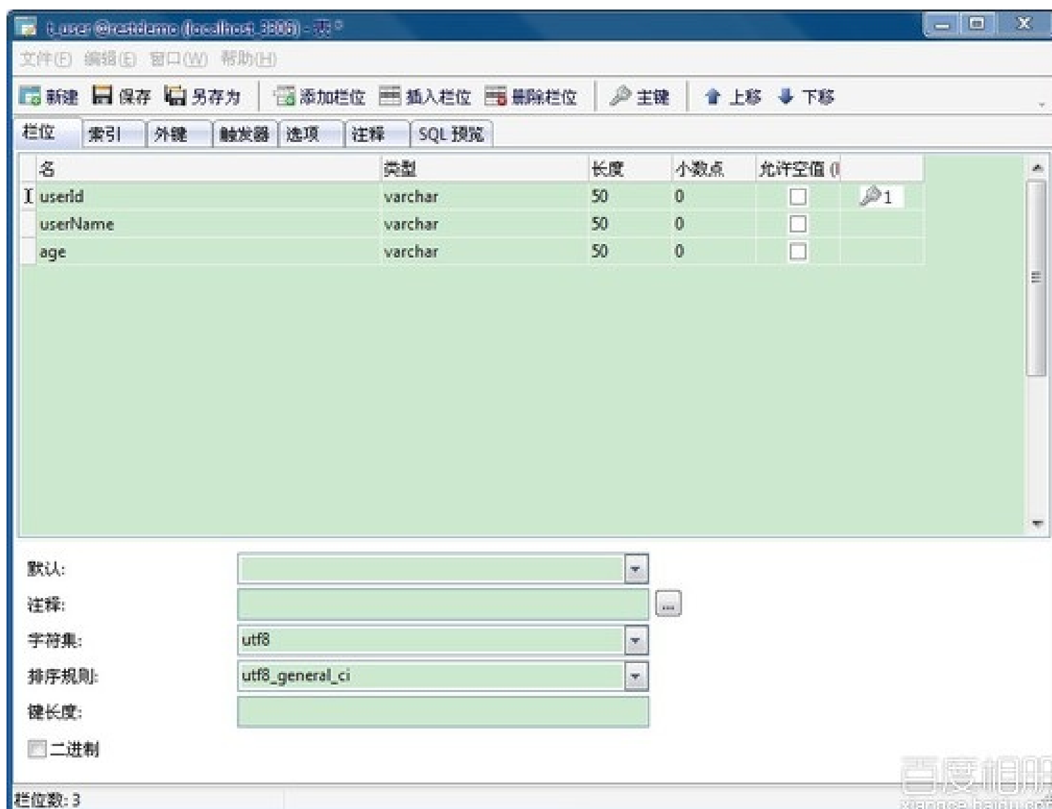
二、环境

1. 上文的项目RestDemo
2. MySQL5.6下载http://dev.mysql.com/get/Downloads/MySQL-5.6/mysql-5.6.16-win32.zip
3. Hibernate4.3.4下载http://sourceforge.net/projects/hibernate/files/hibernate4/4.3.4.Final/hibernate-release-4.3.4.Final.zip
4. Java程序连接MySQL的驱动mysql-connector-java-5.1.29-bin.jar下载 http://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-5.1.29.zip

三、数据库准备

1. 搭建MySQL数据库
2. 创建数据库RestDemo ,及数据表t_user,结构如下

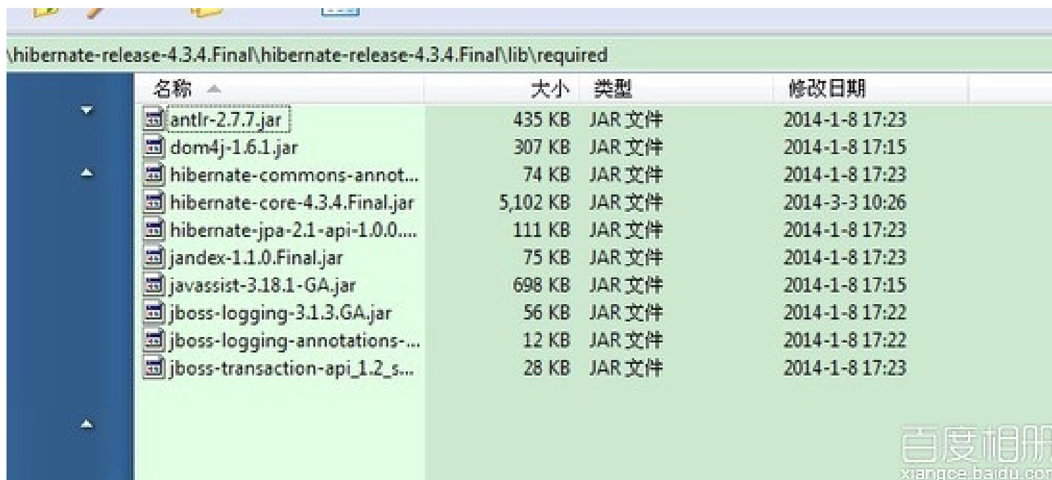
```
DROP TABLE IF EXISTS `t_user`;
CREATE TABLE `t_user` (
  `userId` varchar(50) NOT NULL,
  `userName` varchar(50) NOT NULL,
  `age` varchar(50) NOT NULL,
  PRIMARY KEY (`userId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```



PS: userId 非自增长类型，需要在业务添加

四、引入Hibernate

1. 解压Hibernate的包，在lib\required文件夹下所有jar引入进项目



2. 解压mysql-connector-java-5.1.29.zip，将mysql-connector-java-5.1.29-bin.jar引入进项目

3. 在项目的根目录创建hibernate的配置文件hibernate.cfg.xml，内容如下：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3

<hibernate-configuration>
   <session-factory>
   <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.jdbc
        <property name="connection.url">jdbc:mysql://127.0.0.1:
        <property name="connection.username">root</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDia

        <!-- Enable Hibernate's automatic session context manag
        <property name="current_session_context_class">thread</

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cac

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup
        <property name="hbm2ddl.auto">update</property>

        <mapping resource="com/waylau/rest/bean/User.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

4. 在项目User.java 的同个目录下，创建该类的映射文件User.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtc

<hibernate-mapping package="com.waylau.rest.bean">

    <class name="User" table="T_USER">
        <id name="userId" column="USERID"  type="string" >
            <generator class="assigned"/>
        </id>
        <property name="userName" type="string" />
        <property name="age" type="string" />
    </class>

</hibernate-mapping>
```

5. 创建包com.waylau.rest.util，在该包下创建HibernateUtil.java

```
package com.waylau.rest.util;

import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuild
import org.hibernate.cfg.Configuration;
/**
 * Hibernate 初始化配置工具类
 * @author waylau.com
 * 2014-3-23
 */
public class HibernateUtil {
     private static Configuration configuration;
     private static SessionFactory sessionFactory;
     private static StandardServiceRegistry standardServiceRegi
        static {
            try {
             //第一步:读取Hibernate的配置文件  hibernamte.cfg.xml文
              configuration = new Configuration().configure("hi
             //第二步:创建服务注册构建器对象，通过配置对象中加载所有的配
              StandardServiceRegistryBuilder sb = new StandardS
              sb.applySettings(configuration.getProperties());
             //创建注册服务
              standardServiceRegistry = sb.build();
            //第三步:创建会话工厂
              sessionFactory = configuration.buildSessionFactor
            } catch (Throwable ex) {
                // Make sure you log the exception, as it might
                System.err.println("Initial SessionFactory crea
                throw new ExceptionInInitializerError(ex);
            }
        }

        public static SessionFactory getSessionFactory() {
            return sessionFactory;
        }
}
```

6. 在项目中建com.waylau.rest.dao包，在该包下建立User操作的接口UserDao.java

```
package com.waylau.rest.dao;

import java.util.List;

import com.waylau.rest.bean.User;

/**
 * User Dao 接口
 * @author waylau.com
 * 2014-3-18
 */
public interface UserDao {

    public User getUserById(String id);

    public boolean deleteUserById(String id);

    public boolean createUser(User user);

    public boolean updateUser(User user);

    public List<User> getAllUsers();
}
```

7. 在项目中建com.waylau.rest.dao.impl包，在该包下建立User操作接口的实现UserDaoImpl.java

```
package com.waylau.rest.dao.impl;

import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import com.waylau.rest.bean.User;
import com.waylau.rest.dao.UserDao;
import com.waylau.rest.util.HibernateUtil;
/**
 * 用户DAO实现
 * @author waylau.com
 * 2014-3-23
 */
public class UserDaoImpl implements UserDao {

    @Override
    public User getUserById(String id) {
        SessionFactory sessionFactory = HibernateUtil.getSessio
        Session s = null;
```

```
        Transaction t = null;
        User user = null;
        try{
         s = sessionFactory.openSession();
         t = s.beginTransaction();
         String hql = "from User where userId="+id;
         Query query = s.createQuery(hql);
         user = (User) query.uniqueResult();
         t.commit();
        }catch(Exception err){
        t.rollback();
        err.printStackTrace();
        }finally{
        s.close();
        }
        return user;
    }

    @Override
    public boolean deleteUserById(String id) {
        SessionFactory sessionFactory = HibernateUtil.getSessic
        Session s = null;
        Transaction t = null;
        boolean flag = false;
        try{
         s = sessionFactory.openSession();
         t = s.beginTransaction();
         User user = new User();
         user.setUserId(id);
         s.delete(user);
         t.commit();
         flag = true;
        }catch(Exception err){
        t.rollback();
        err.printStackTrace();
        }finally{
        s.close();
        }
        return flag;
    }

    @Override
    public boolean createUser(User user) {
        SessionFactory sessionFactory = HibernateUtil.getSessic
        Session s = null;
        Transaction t = null;
        boolean flag = false;
        try{
         s = sessionFactory.openSession();
         t = s.beginTransaction();
         s.save(user);
         t.commit();
         flag = true;
```

```
        }catch(Exception err){
        t.rollback();
        err.printStackTrace();
        }finally{
        s.close();
        }
        return flag;
    }

    @Override
    public boolean updateUser(User user) {
        SessionFactory sessionFactory = HibernateUtil.getSessic
        Session s = null;
        Transaction t = null;
        boolean flag = false;
        try{
         s = sessionFactory.openSession();
         t = s.beginTransaction();
         s.update(user);
         t.commit();
         flag = true;
        }catch(Exception err){
        t.rollback();
        err.printStackTrace();
        }finally{
        s.close();
        }
        return flag;
    }

    @Override
    public List<User> getAllUsers() {
        SessionFactory sessionFactory = HibernateUtil.getSessic
        Session s = null;
        Transaction t = null;
        List<User> uesrs = null;
        try{
         s = sessionFactory.openSession();
         t = s.beginTransaction();
         String hql = "select * from t_user";
         Query query = s.createSQLQuery(hql).addEntity(User.cla
         query.setCacheable(true); // 设置缓存
         uesrs = query.list();
         t.commit();
        }catch(Exception err){
        t.rollback();
        err.printStackTrace();
        }finally{
        s.close();
        }
        return uesrs;
    }
```

```
}
```

8. 修改项目中 com.waylau.rest.resources包下的UserResource.java，使之前在内存中模拟CURD转为在数据库中实现

```
package com.waylau.rest.resources;

import java.util.ArrayList;
import java.util.List;

import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.Consumes;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;

import com.waylau.rest.bean.User;
import com.waylau.rest.dao.impl.UserDaoImpl;

/**
 * 用户资源
 * @author waylau.com
 * 2014-3-19
 */
@Path("/users")
public class UserResource {
    private UserDaoImpl userDaoImpl = new UserDaoImpl();
    /**
     * 增加
     * @param user
     */
    @POST
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION
    public void createUser(User user) {
        userDaoImpl.createUser(user);
    }

    /**
     * 删除
     * @param id
     */
    @DELETE
    @Path("{id}")
    public void deleteUser(@PathParam("id")String id){
        userDaoImpl.deleteUserById(id);
    }
```

```
    /**
     * 修改
     * @param user
     */
    @PUT
    @Consumes(MediaType.APPLICATION_XML)
    public void updateUser(User user){
        userDaoImpl.updateUser(user);
    }

    /**
     * 根据id查询
     * @param id
     * @return
     */
    @GET
    @Path("{id}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION
    public User getUserById(@PathParam("id") String id){
        User u = userDaoImpl.getUserById(id);
        return u;
    }

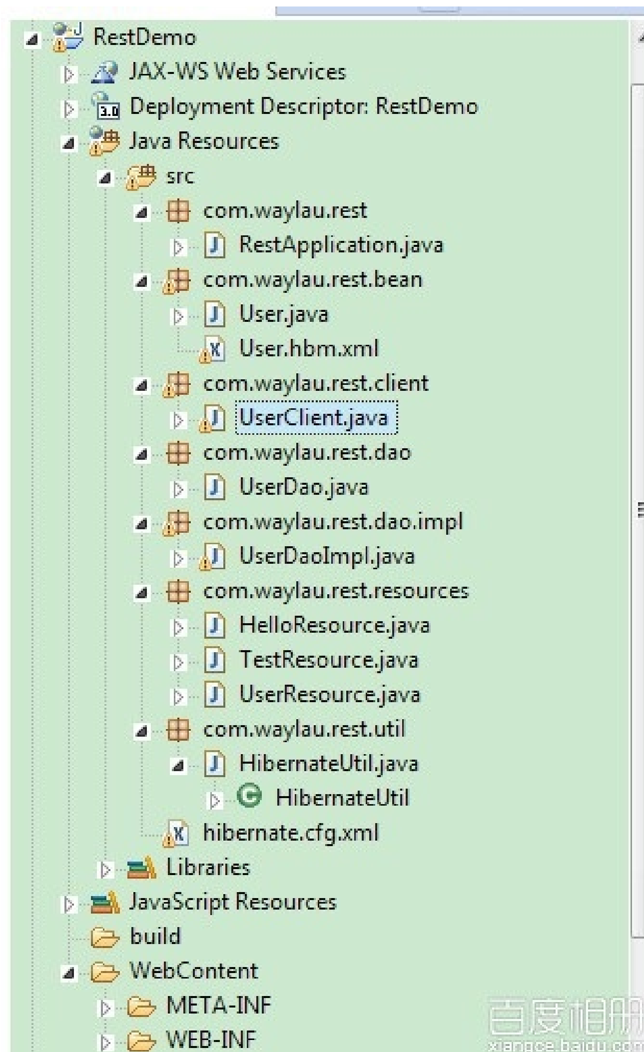
    /**
     * 查询所有
     * @return
     */
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION
    public List<User> getAllUsers(){
        List<User> users = new ArrayList<User>();
        users = userDaoImpl.getAllUsers();
        return users;
    }

}
```

五、运行

1. 将服务端运行后
2. 运行UserClient客户端，可以看到数据库已经实现增删改查

完整项目架构如下：



本章源码：https://github.com/waylau/RestDemo/tree/master/jersey-demo5-mysql-hibernate

用Jersey构建RESTful服务6--Jersey+SQLServer+Hibernate4.3

一、总体说明

本例运行演示了用Jersey构建RESTful服务中，如何同过Hibernate将数据持久化进SQLServer的过程

二、环境

1. 上文的项目RestDemo
2. SQLServer2005
3. jtds数据库连接驱动：下载地址最新版本,替换掉上文项目中的 `mysql-connector`

三、配置

1. 与上文mysql的配置不同点主要在hibernate.cfg.xml文件； 配置如下：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3

<hibernate-configuration>
   <session-factory>
   <!-- Database connection settings -->
        <property name="connection.driver_class">net.sourceforg
        <property name="connection.url">jdbc:jtds:sqlserver://1
        <property name="connection.username">sa</property>
        <property name="connection.password">aA123456</property
        <property name="hibernate.default_schema">RestDemo</pro
        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>
        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.SQLServe
        <!-- Enable Hibernate's automatic session context manag
        <property name="current_session_context_class">thread</
        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cac
        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>
        <!-- Drop and re-create the database schema on startup
        <property name="hbm2ddl.auto">update</property>
        <mapping resource="com/waylau/rest/bean/User.hbm.xml"/>
      </session-factory>
</hibernate-configuration>
```

2. 修改于mysql不兼容的sql语句 `com.waylau.rest.dao.impl` 中的 `UserDaoImpl`：

   getUserById修改成如下：

```
@Override
public User getUserById(String id) {
    SessionFactory sessionFactory = HibernateUtil.getSessionFac
    Session s = null;
    Transaction t = null;
    User user = null;
    try{
     s = sessionFactory.openSession();
     t = s.beginTransaction();
     String hql = "from User where userId='"+id+"'";
     Query query = s.createQuery(hql);
     user = (User) query.uniqueResult();
     t.commit();
    }catch(Exception err){
    t.rollback();
    err.printStackTrace();
    }finally{
    s.close();
    }
    return user;
}
```

getAllUsers给成如下：

```
@Override
public List<User> getAllUsers() {
    SessionFactory sessionFactory = HibernateUtil.getSessionFac
    Session s = null;
    Transaction t = null;
    List<User> uesrs = null;
    try{
     s = sessionFactory.openSession();
     t = s.beginTransaction();
     String hql = "select * from [RestDemo].dbo.t_user";
     Query query = s.createSQLQuery(hql).addEntity(User.class);
     //query.setCacheable(true); // 设置缓存
     uesrs = query.list();
     t.commit();
    }catch(Exception err){
    t.rollback();
    err.printStackTrace();
    }finally{
    s.close();
    }
    return uesrs;
}
```

或者如下：

```
@Override
public List<User> getAllUsers() {
    SessionFactory sessionFactory = HibernateUtil.getSessionFac
    Session s = null;
    Transaction t = null;
    List<User> uesrs = null;
    try{
     s = sessionFactory.openSession();
     t = s.beginTransaction();
     String hql = " from User";
     Query query = s.createQuery(hql);
     //query.setCacheable(true); // 设置缓存
     uesrs = query.list();
     t.commit();
    }catch(Exception err){
    t.rollback();
    err.printStackTrace();
    }finally{
    s.close();
    }
    return uesrs;
}
```

四、问题

可能会出现如下错误

```
ERROR: 指定的架构名称 "RestDemo" 不存在，或者您没有使用该名称的权限。
三月 26, 2014 3:38:43 下午 org.hibernate.tool.hbm2ddl.SchemaUpdate e
INFO: HHH000232: Schema update complete
Hibernate: insert into RestDemo.T_USER (userName, age, USERID) valu
三月 26, 2014 3:38:43 下午 org.hibernate.engine.jdbc.spi.SqlExceptio
WARN: SQL Error: 208, SQLState: S0002
三月 26, 2014 3:38:43 下午 org.hibernate.engine.jdbc.spi.SqlExceptio
ERROR: 对象名 'RestDemo.T_USER' 无效。
```

解决方案：

将配置文件中的 `hibernate.default_schema` 值修改为如下即可：

```
<property name="hibernate.default_schema">RestDemo.dbo</property>
```

或者去掉上面的配置，在“User.hbm.xml”修改如下

```
<class name="User" table="T_USER" schema="RestDemo.dbo">
```

本章源码：https://github.com/waylau/RestDemo/tree/master/jersey-demo6-sqlserver-hibernate

https://github.com/waylau/RestDemo/tree/master/jersey-demo6.2-sqlserver-hibernate

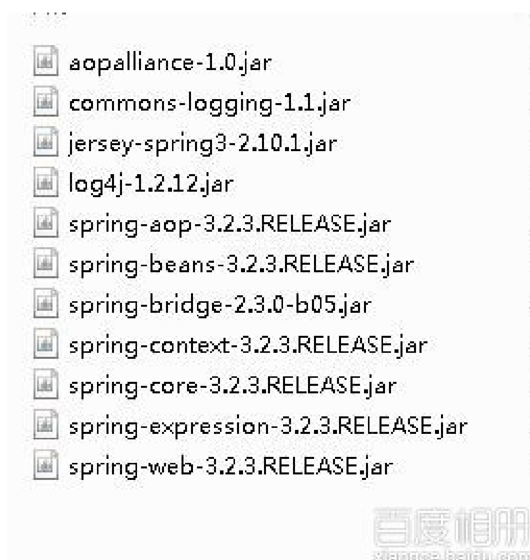
用Jersey构建RESTful服务7--Jersey+SQLServer+Hibernate4.3+Spring3.2

一、总体说明

本例运行演示了用 Jersey 构建 RESTful 服务中，如何集成 Spring3

二、环境

1. 上文的项目RestDemo
2. Spring及其他相关的jar ,导入项目



三、配置

1. 根目录下下创建 Spring 的配置文件 `applicationContext.xml` ；配置如下： <?xml version="1.0" encoding="UTF-8"?>

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/cor
       xsi:schemaLocation="http://www.springframework.org/schem
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.
        >
    <!-- 激活那些已经在g容器里注册过的bean -->
    <context:annotation-config></context:annotation-config>

    <!-- 在容器中注入bean -->
    <bean id="UserServiceImpl" class="com.waylau.rest.service.i
    <bean id="UserDaoImpl" class="com.waylau.rest.dao.impl.User

</beans>
```

2. 在 `com.waylau.rest.service` 和 `com.waylau.rest.service.impl` 下分别增加 UserService 和 UserServiceImpl。

`UserService.java`

```
package com.waylau.rest.service;

import java.util.List;

import com.waylau.rest.bean.User;

/**
 * User Service 接口
 * @author waylau.com
 * 2014-7-25
 */
public interface UserService {

    public User getUserById(String id);

    public boolean deleteUserById(String id);

    public boolean createUser(User user);

    public boolean updateUser(User user);

    public List<User> getAllUsers();
}
```

`UserServiceImpl.java`

```
package com.waylau.rest.service.impl;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import com.waylau.rest.bean.User;
import com.waylau.rest.dao.impl.UserDaoImpl;
import com.waylau.rest.service.UserService;
/**
 * User Service 接口实现
 * @author waylau.com
 * 2014-7-25
 */
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDaoImpl userDaoImpl;

    public UserServiceImpl() {
        // TODO Auto-generated constructor stub
    }

    @Override
    public User getUserById(String id) {
        return userDaoImpl.getUserById(id);
    }

    @Override
    public boolean deleteUserById(String id) {
        return userDaoImpl.deleteUserById(id);
    }

    @Override
    public boolean createUser(User user) {
        return userDaoImpl.createUser(user);
    }

    @Override
    public boolean updateUser(User user) {
        return userDaoImpl.updateUser(user);
    }

    @Override
    public List<User> getAllUsers() {
        return userDaoImpl.getAllUsers();
    }

}
```

3. 修改 `UserResource.java`

```
package com.waylau.rest.resources;

import java.util.ArrayList;
import java.util.List;
import java.util.logging.Logger;

import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.Consumes;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;

import org.springframework.beans.factory.annotation.Autowired;

import com.waylau.rest.bean.User;
import com.waylau.rest.service.impl.UserServiceImpl;

/**
 * 用户资源
 * @author waylau.com
 * 2014-7-26
 */
@Path("/users")
public class UserResource {
    private static final Logger LOGGER = Logger.getLogger(UserR

    @Autowired
    private UserServiceImpl userServiceImpl;

    public UserResource() {
        LOGGER.fine("UserResource()");
    }

    /**
     * 增加
     * @param user
     */
    @POST
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION
    public void createUser(User user) {
        userServiceImpl.createUser(user);
    }

    /**
     * 删除
     * @param id
```

```
     */
    @DELETE
    @Path("{id}")
    public void deleteUser(@PathParam("id")String id){
        userServiceImpl.deleteUserById(id);
    }

    /**
     * 修改
     * @param user
     */
    @PUT
    @Consumes(MediaType.APPLICATION_XML)
    public void updateUser(User user){
        userServiceImpl.updateUser(user);
    }

    /**
     * 根据id查询
     * @param id
     * @return
     */
    @GET
    @Path("{id}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION
    public User getUserById(@PathParam("id") String id){
        User u = userServiceImpl.getUserById(id);
        return u;
    }

    /**
     * 查询所有
     * @return
     */
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION
    public List<User> getAllUsers(){
        List<User> users = new ArrayList<User>();
        users = userServiceImpl.getAllUsers();
        return users;
    }
}
```

4. 修改web.xml,插入

```
<module-name>RestDemo</module-name>

<listener>
  <listener-class>org.springframework.web.context.ContextLoader
</listener>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext.xml</param-value>
</context-param>
```

四、运行

1. 先运行项目
2. 运行UserClient.java测试，控制台输出如下

```
****增加用户addUser****
****查询所有getAllUsers****
[{"userId":"002","userName":"sdfs","age":"23"},{"userId":"003",
****修改用户updateUser****
****根据id查询用户****
006Susan33
****查询所有getAllUsers****
[{"userId":"002","userName":"sdfs","age":"23"},{"userId":"003",
****删除用户****
****查询所有getAllUsers****
[{"userId":"002","userName":"sdfs","age":"23"},{"userId":"003",
```

本章源码（含**jar**包）：https://github.com/waylau/RestDemo/tree/master/jersey-demo7-sqlserver-hibernate-spring3

用Jersey构建RESTful服务8--Jersey+SQLServer+Hibernate4.3+Spring3.2+jquery

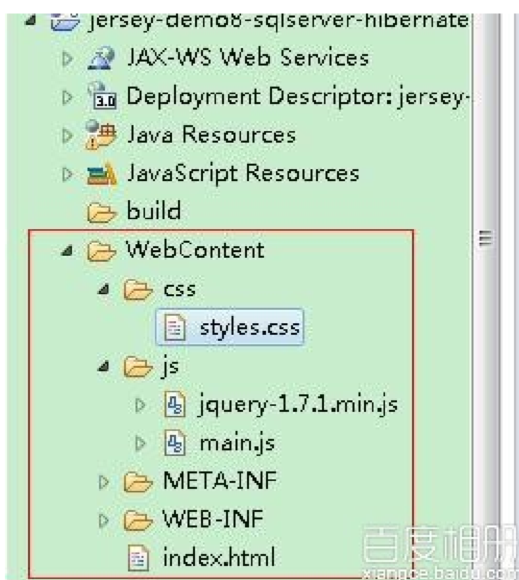
一、总体说明

本例运行演示了用 Jersey 构建 RESTful 服务中，如何集成 jQuery,用html作为客户端访问 RESTful 服务。

二、环境

1. 上文的项目RestDemo
2. jQuery 库 ,本例为1.7.1版本

三、配置



1. 创建 jQuery 客户端的项目结构，在 `WebContent` 创建 `js` , `css` 两个目录，并把jQuery 库 放入 `js` 目录下，并在该目录下创建 `main,js` 空文件
2. 在 `WebContent` 创建 `index.html` :

```
<!DOCTYPE HTML>
   <html>
   <head>
   <title>jquery Demo (人员管理系统)</title>
   <meta charset="utf-8"/>
   <link rel="stylesheet" href="css/styles.css" />

   <script src="js/jquery-1.7.1.min.js"></script>
   <script src="js/main.js"></script>
   </head>

   <body>
       <div class="header">

           <button id="btnClear">Clear</button>
           <button id="btnRefreash">Refreash</button>
           更多实例访问：<a href="http://www.waylau.com" >www.wa
       </div>

       <div class="leftArea">
           <ul id="userList"></ul>
       </div>

       <form id="wineForm">

           <div class="mainArea">

               <label>Id:</label> <input id="userId" name="use

               <label>Name:</label> <input type="text" id="use
               <label>Age:</label> <input  type="text" id="age

               <button id="btnAdd">Add</button>
               <button id="btnSave">Save</button>
               <button id="btnDelete">Delete</button>
           </div>

       </form>

   </body>
   </html>
```

3. 修改 `main.js`

```
// The root URL for the RESTful services
   var rootURL = 'http://localhost:8089/RestDemo/rest/users';

   var currentUser;

   // Retrieve wine list when application starts
```

```
    findAll();

    // Nothing to delete in initial application state
    $('#btnDelete').hide();

    $('#btnAdd').click(function() {
        addUser();
        return false;
    });

    $('#btnSave').click(function() {

        updateUser();
        return false;
    });

    $('#btnClear').click(function() {
        newUser();
        return false;
    });

    $('#btnDelete').click(function() {
        deleteUser();
        return false;
    });

    $('#userList a').live('click', function() {
        findById($(this).data('identity'));
    });

    $('#btnRefreash').click(function() {
        findAll();
        return false;
    });

    function newUser() {
        $('#btnDelete').hide();
        currentUser = {};
        renderDetails(currentUser); // Display empty form
    }

    function findAll() {
        console.log('findAll');
        $.ajax({
            type: 'GET',
            url: rootURL,
            dataType: "json", // data type of response
            success: renderList
        });
    }

    function findById(id) {
        console.log('findById: ' + id);
```

```
        $.ajax({
            type: 'GET',
            url: rootURL + '/' + id,
            dataType: "json",
            success: function(data){
                $('#btnDelete').show();

                console.log('findById success: ' + data.userNam
                currentUser = data;
                renderDetails(currentUser);
            }

        });
    }

    function addUser() {
        console.log('addUser');
        $.ajax({
            type: 'POST',
            contentType: 'application/json',
            url: rootURL,
            dataType: "json",
            data: formToJSON(),
            success: function(data, textStatus, jqXHR){
                alert('User created successfully');
                $('#btnDelete').show();
                $('#userId').val(data.userId);
            },
            error: function(jqXHR, textStatus, errorThrown){
                alert('addUser error: ' + textStatus);
            }
        });
    }

    function updateUser() {
        console.log('updateUser');
        $.ajax({
            type: 'PUT',

            contentType: 'application/json',
            url: rootURL,
            dataType: "json",
            data: formToJSON(),

            success: function(data, textStatus, jqXHR){
                alert('User updated successfully');
            },
            error: function(jqXHR, textStatus, errorThrown){
                alert('updateUser error: ' + textStatus);
            }
        });
    }
```

```
    function deleteUser() {
        console.log('deleteUser');
        $.ajax({
            type: 'DELETE',
            url: rootURL + '/' + $('#userId').val(),
            success: function(data, textStatus, jqXHR){
                alert('user deleted successfully');
            },
            error: function(jqXHR, textStatus, errorThrown){
                alert('delete user error');
            }
        });
    }

    function renderList(data) {
        // JAX-RS serializes an empty list as null, and a 'coll
        var list = data == null ? [] : (data instanceof Array ?

        $('#userList li').remove();
        $.each(list, function(index, data) {
            $('#userList').append('<li><a href="#" data-identit
        });
    }

    function renderDetails(data) {
        $('#userId').val(data.userId);
        $('#userName').val(data.userName);
        $('#age').val(data.age);

    }

    // Helper function to serialize all the form fields into a
    function formToJSON() {
        var userId = $('#userId').val();
        return JSON.stringify({
            "userId": userId == "" ? null : userId,
            "userName": $('#userName').val(),
            "age": $('#age').val()
            });
    }
```

4. 在 `css` 目录下创建 `styles.css` 文件

```
  * {
      font-family: "Helvetica Neue", Helvetica, Arial, sans-ser
      font-size: 18px;
    }

    .header {
        padding-top: 5px;
```

```
    }

    .leftArea {
        position: absolute;
        left: 10px;
        top: 70px;
        bottom: 20px;
        width: 260px;
        border:solid 1px #CCCCCC;
        overflow-y: scroll;
    }

    .mainArea {
        position: absolute;
        top: 70px;
        bottom: 20px;
        left:300px;
        overflow-y: scroll;
        width:300px;
    }

    .rightArea {
        position: absolute;
        top: 70px;
        bottom: 20px;
        left:650px;
        overflow-y: scroll;
        width:270px;
    }

    ul {
        list-style-type: none;
        padding-left: 0px;
        margin-top: 0px;
    }

    li a {
        text-decoration:none;
        display: block;
        color: #000000;
        border-bottom:solid 1px #CCCCCC;
        padding: 8px;
    }

    li a:hover {
        background-color: #4B0A1E;
        color: #BA8A92;
    }

    input, textarea {
      border:1px solid #ccc;
      min-height:30px;
```

```
        outline: none;
      }

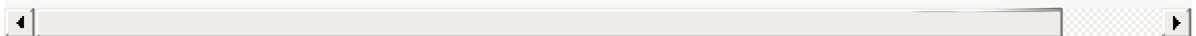
      .mainArea input {
        margin-bottom:15px;
        margin-top:5px;
        width:280px;
      }

      textarea {
          margin-bottom:15px;
          margin-top:5px;
          height: 200px;
          width:250px;
      }

      label {
          display:block;
      }

      button {
          padding:6px;
      }

      #searchKey {
          width:160px;
      }
```



四、运行

1. 先运行项目
2. 可以进行CURD操作

PS:本案例力求简单把jquery访问 RESTful 服务展示出来，代码只在Chrome上做过测试。

本章源码：https://github.com/waylau/RestDemo/tree/master/jersey-demo8-sqlserver-hibernate-spring3-jquery

用Jersey构建RESTful服务9--Jersey+SQLServer+Hibernate4.3+Spring3.2+AngularJS

一、总体说明

本例运行演示了用 Jersey 构建 RESTful 服务中，如何集成 angular,用MVC分层的方式访问 RESTful 服务。

二、环境

1. 上上文的项目 Demo7)
2. angular 库 ,本例为1.2.3 版本
3. 样式 bootstrap-3.1.1.min.js

三、配置

1. 完成项目结构



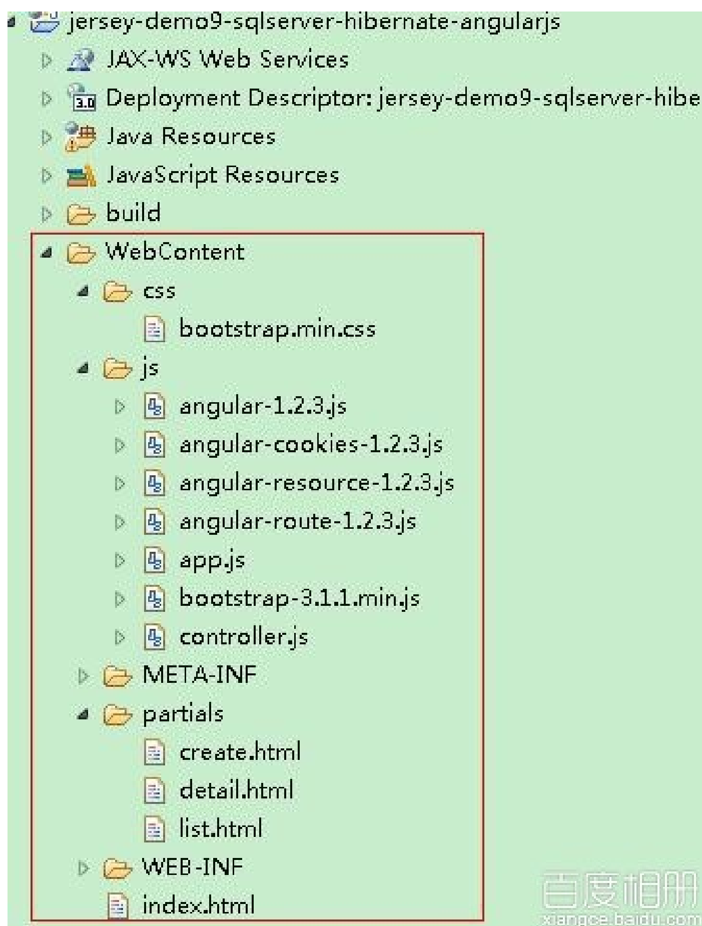
创建相应的目录结构

angularjs 、bootstrap 的js,css文件放别放入相应的目录，

在js目录下再创建 `app.js` 、 `controller.js`

在partials目录下再创建 `create.html` 、 `list.html` 、 `detail.html`

完整目录结构如下



2. 在 `list.html` 填入如下内容,主要是显示用户列表 `ng-repeat` 为 angularjs 迭代器 作用是数据绑定:

```
<div class="pull-right">
        <a href="#/create" class="btn btn-default" title="Creat
    </div>
    <div class="page-header">
        <h3>Users</h3>
    </div>
    <hr />
    <li ng-repeat="user in users | filter:query | orderBy:order

        <div class="pull-right">
            <a href="#/users/"  class="btn btn-xs btn-default" t
            <span class="glyphicon glyphicon-pencil"></span></a>
        </div>
        <h4>userId: </h4>
        <p>userName:<a href="#/users/"></a>   Age:</p>
    <hr />
    </li>

    <hr />
```

3. 修改 `create.html` 用来添加用户信息, `ng-model` 是模型

```
<div class="page-header">
        <h3>Create</h3>
    </div>

    <form role="form" name="userForm">

     <div class="row"> </div>
     <div class="row" ng-class="{'has-error': userForm.userId.$
      <div class="col-md-1">
        <i ng-show="userForm.url.$error.required"
          class="glyphicon glyphicon-pencil"></i>
        <label for="urlInput">userId</label>
      </div>
      <div class="col-md-4">
        <input
          type="test"
          class="form-control" id="urlInput"
          name="userId" ng-model="user.userId" required>
      </div>
    </div>
    <div class="row"> </div>

    <div class="row" ng-class="{'has-error': userForm.userName.
      <div class="col-md-1">
        <i ng-show="userForm.userName.$error.required"
          class="glyphicon glyphicon-pencil"></i>
        <label for="nameInput">userName</label>
      </div>
      <div class="col-md-4">
        <input
          type="text"
          class="form-control error" id="nameInput"
          name="userName"
          ng-model="user.userName"
          required>
      </div>
    </div>

    <div class="row"> </div>
    <div class="row" ng-class="{'has-error': userForm.url.$inva
      <div class="col-md-1">
        <i ng-show="userForm.url.$error.required"
          class="glyphicon glyphicon-pencil"></i>
        <label for="urlInput">age</label>
      </div>
      <div class="col-md-4">
        <input
          type="text"
          class="form-control" id="urlInput"
          name="age" ng-model="user.age" required>
      </div>
    </div>
```

```
    <div class="row" ng-hide="showConfirm">
      <div class="col-md-5">
        <a href="#users" class="btn">Cancel</a>
        <button
          ng-click="add()"
          ng-disabled="isClean() || userForm.$invalid"
          class="btn btn-primary">Save</button>

      </div>
    </div>
```

4. 修改 `detail.html` 用来显示用户信息并提供修改、删除等功能

```
<form role="form" name="userForm">

     <div class="row"> </div>
     <div class="row" ng-class="{'has-error': userForm.userId.$
      <div class="col-md-1">
        <i ng-show="userForm.url.$error.required"
          class="glyphicon glyphicon-pencil"></i>
        <label for="urlInput">userId</label>
      </div>
      <div class="col-md-4">
        <input
          type="test"
          class="form-control" id="urlInput"
          name="userId" ng-model="user.userId" required>
      </div>
    </div>
    <div class="row"> </div>

    <div class="row" ng-class="{'has-error': userForm.userName.
      <div class="col-md-1">
        <i ng-show="userForm.userName.$error.required"
          class="glyphicon glyphicon-pencil"></i>
        <label for="nameInput">userName</label>
      </div>
      <div class="col-md-4">
        <input
          type="text"
          class="form-control error" id="nameInput"
          name="userName"
          ng-model="user.userName"
          fend-focus="focusUserNameeInput"
          required>
      </div>
    </div>
    <div class="row"> </div>
    <div class="row" ng-class="{'has-error': userForm.url.$inva
```

```
        <div class="col-md-1">
          <i ng-show="userForm.url.$error.required"
            class="glyphicon glyphicon-pencil"></i>
          <label for="urlInput">age</label>
        </div>
        <div class="col-md-4">
          <input
            type="text"
            class="form-control" id="urlInput"
            name="age" ng-model="user.age" required>
        </div>
      </div>

      <div class="row" ng-hide="showConfirm"> </div>

      <div class="row" ng-hide="showConfirm">
        <div class="col-md-5">
          <a href="#users" class="btn">Cancel</a>
          <button
            ng-click="save()"
            ng-disabled="isClean() || userForm.$invalid"
            class="btn btn-primary">Save</button>
          <button
            ng-click="remove()"
            ng-show="user.userId"
            class="btn btn-danger">Delete</button>
        </div>
      </div>

    </form>
```

5. 修改 `index.html` 作为主页面,嵌入其他子页面， `ng-app` 声明这个是模块， `ng-controller` 说明他的控制器叫 `ListCtrl` , `ng-view` 用来存放子视图（页面）。

```
<!doctype html>
   <html ng-app="appMain" ng-controller="ListCtrl">
   <head>
       <meta charset="utf-8"/>
       <meta name="viewport" content="width=device-width, init
       <link rel="stylesheet" href="css/bootstrap.min.css" typ
   </head>
   <body >
       <!-- navbar -->
       <div class="container ng-view"></div>
       <!-- footer -->
       <div id="footer" class="hidden-xs">
         <div class="container">
           <p class="text-muted">
             Project Example - by <a href="http://www.waylau.c
             <a href="https://github.com/waylau" target="_blar
           </p>
         </div>
       </div>

       <script src="js/bootstrap-3.1.1.min.js"></script>
       <script src="js/angular-1.2.3.js"></script>
       <script src="js/angular-resource-1.2.3.js"></script>
       <script src="js/angular-route-1.2.3.js"></script>
       <script src="js/angular-cookies-1.2.3.js"></script>

       <script src="js/app.js"></script>
       <script src="js/controller.js"></script>
   </body>
   </html>
```

6. 修改 `app.js` ，声明模块 `appMain` ,提供路由功能，说明了调转到哪个页面，用哪个控制器

```
angular.module('appMain', ['ngRoute']).config(['$routeProvider
       $routeProvider.
           when('/users', {templateUrl: 'partials/list.html',
           when('/users/:userId', {templateUrl: 'partials/deta
           when('/create', {
               templateUrl: 'partials/create.html',
               controller: CreateController
           }).
           otherwise({redirectTo: '/users'});
   }]);
```

7. 修改 `controller.js` ，控制器。主要是对业务逻辑的操作，常见的CURD功能，http访问RESTful接口，并且返回数据

```
var url = 'http://localhost:8089/RestDemo/rest';

    function ListCtrl($scope, $http) {
        $http.get( url + '/users' ).success(function(data) {
            $scope.users = data;
        });

        $scope.orderProp = 'age';
    }

    function DetailCtrl($scope, $routeParams, $http) {

        $http.get( url + '/users/'+$routeParams.userId).success
            $scope.user = data;
        });

        $scope.save = function() {
            $http.put( url + '/users',  $scope.user).
            success(function(data, status, headers, config){
                $location.path('/');
            }).error(function(data, status, headers, config){
                alert("error"+status);
            }) ;
        };

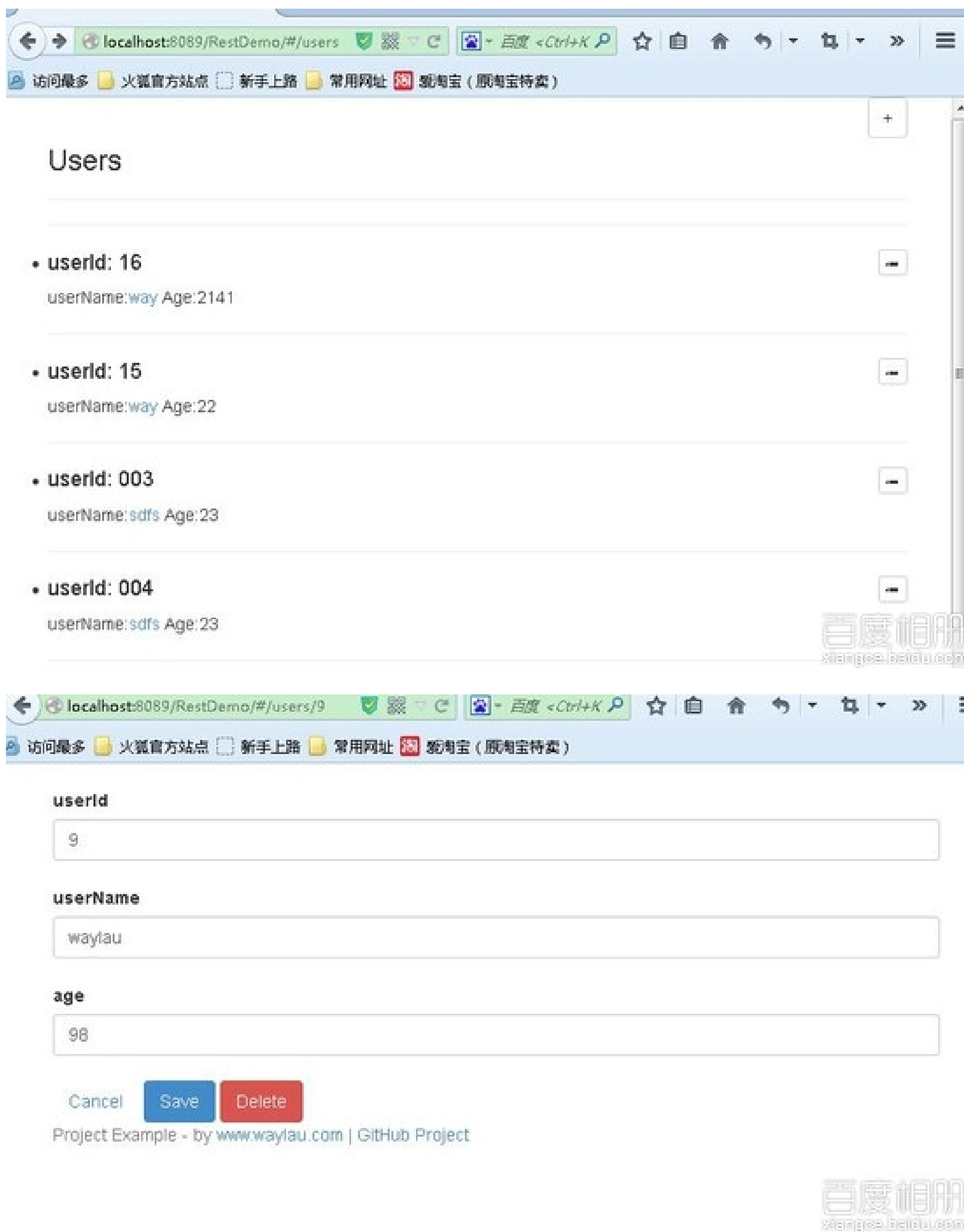
        $scope.remove =  function(){
            $http({
                method:'DELETE',
                url: url + '/users/'+  $scope.user.userId ,
            })
            .success(function(data, status, headers, config){
                $location.path('/');
            }).error(function(data, status, headers, config){
                alert("error"+status);
            }) ;
        };
    }

    function CreateController($scope, $http) {

        $scope.add = function() {
            $http.post( url + '/users',  $scope.user).
            success(function(data, status, headers, config){
                $location.path('/');
            }).error(function(data, status, headers, config){
                alert("error"+status);
            }) ;
        };
    }
```

四、运行

1. 先运行项目
2. 可以进行CURD操作



PS:本案例力求简单把 angularjs 访问 RESTful 服务展示出来，在Chrome,firefox,IE上做过测试。

本章源码：jersey-demo9-sqlserver-hibernate-spring3-angularjs

用 Jersey 2 和 Spring 4 构建 RESTful web service

本文介绍了如何通过 Jersey 框架优美的在 Java 实现了 REST 的 API。CRUD 的操作存储在 MySQL 中

1. 示例

1.1 为什么

Spring 可以对于 REST 有自己的实现(见 https://spring.io/guides/tutorials/rest/)。但本文展示的是用 “官方” 的 方法来实现 REST ，即使用 Jersey。

1.2 它是做什么的?

管理 资源。 REST API 将允许创建、检索、更新和删除这样的资源。

1.3 架构及技术

本示例项目使用多层结构，基于“Law of Demeter (LoD) or principle of least knowledge”（迪米特法则），是说一个软件实体要尽可能的只与和它最近的实体进行通讯。通常被表述为：talk only to your immediate friends (只和离你最近的朋友进行交互)。 “talk”，其实就是对象间方法的调用。这条规则表明了对象间方法调用的原则：（1）调用对象本身的方法；（2）调用通过参数传入的对象的方法；（3）在方法中创建的对象的方法；（4）所包含对象的方法。

主要分为三层：

- 第一层：Jersey 实现对 REST 的支持，拥有 外观模式的角色并代理到逻辑业务层
- 业务层: 发生逻辑的地方
- 数据访问层： 是与持久数据存储（在我们的例子中是 MySql数据库)交互的地方

简述下技术框架：

1.3.1. Jersey (外观)

Jersey 是开源、拥有产品级别的质量，提供构建 RESTful Web Services, 支持 JAX-RS APIs ，提供 JAX-RS (JSR 311 & JSR 339) 参考实现。

1.3.2. Spring (业务层)

在我看来没有什么 比 Spring 更好的办法让 pojo 具有不同的功能。 你会发现在本教程用 Jersey 2 和 Spring 4 构建 RESTful web service

1.3.3. JPA 2 / Hibernate (持久层)

使用 Hibernate 实现 DAO 模式。

1.3.4. Web 容器

用 Maven 打包成 .war 文件开源部署在任意容器。一般用 Tomcat 和 Jetty ，也可以是 Glassfih, Weblogic, JBoss 或 WebSphere.

1.3.5. MySQL 数据库

示例数据存储在一个 MySQL 表:

1.3.6. 技术版本

Jersey 2.9

Spring 4.0.3

Hibernate 4

Maven 3

Tomcat 7

Jetty 9

MySql 5.6

1.4. 源码

见https://github.com/waylau/RestDemo/tree/master/jersey-2-spring-4-rest

2. 配置

开始呈现 REST API 的设计和实现之前,我们需要做一些配置。

2.1. 项目依赖

Jersey Spring 扩展包 是必须要放在 项目 classpath 中。在 pom.xml 中添加下面依赖：

```
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-spring3</artifactId>
    <version>${jersey.version}</version>
    <exclusions>
        <exclusion>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.4.1</version>
</dependency>
```

注意*: jersey-spring3.jar* 使用的是他自己的 *Spring* 库版本，所以如果你想使用自己的 *(*本例是使用 *Spring 4.0.3.Release),*你需要将这些库手动的移除。如果想看到其他 的库的依赖，请查看项目源码中的 *pom.xml*

2.2. web.xml

应用部署描述

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://ja
    <display-name>Demo - Restful Web Application</display-name>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring/applicationContext.xml</paran
    </context-param>

    <servlet>
        <servlet-name>jersey-serlvet</servlet-name>
        <servlet-class>
            org.glassfish.jersey.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>org.codingpedia.demo.rest.RestDemoJaxRsApp
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>jersey-serlvet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <resource-ref>
        <description>Database resource rest demo web application </
        <res-ref-name>jdbc/restDemoDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</web-app>
```

2.2.1. Jersey-servlet

注意 Jersey servlet 的配置， `javax.ws.rs.core.Application` 类定义了 JAX-RS 应用组件(root 资源 和 提供者 类) .本例使用 `ResourceConfig` , 是 Jersey 自己实现的 `Application` 类，提供了简化 JAX-RS 组件的能力。详见JAX-RS 应用模型

`org.codingpedia.demo.rest.RestDemoJaxRsApplication` 是自己实现的 `ResourceConfig` 类，注册应用的 resources, filters, exception mappers 和 feature :

```
package org.codingpedia.demo.rest.service;

//imports omitted for brevity

/**
 * Registers the components to be used by the JAX-RS application
 *
 * @author ama
 *
 */
public class RestDemoJaxRsApplication extends ResourceConfig {

    /**
     * Register JAX-RS application components.
     */
    public RestDemoJaxRsApplication() {
        // register application resources
        register(PodcastResource.class);
        register(PodcastLegacyResource.class);

        // register filters
        register(RequestContextFilter.class);
        register(LoggingResponseFilter.class);
        register(CORSResponseFilter.class);

        // register exception mappers
        register(GenericExceptionMapper.class);
        register(AppExceptionMapper.class);
        register(NotFoundExceptionMapper.class);

        // register features
        register(JacksonFeature.class);
        register(MultiPartFeature.class);
    }
}
```

注意：

- `org.glassfish.jersey.server.spring.scope.RequestContextFilter` 是 Spring filter 提供了 JAX-RS 和 Spring 请求属性之间的桥梁。
- `org.codingpedia.demo.rest.resource.PodcastsResource` 这是“外观”组件，通过注解 暴露了 REST 的API。稍后会描述
- `org.glassfish.jersey.jackson.JacksonFeature` ,是一个 feature ，用 Jackson JSON 的提供者来解释 JSON。

2.1.2. Spring 配置

配置文件在 classpath 目录下的 spring/applicationContext.xml:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xs

        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd

        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-contex

    <context:component-scan base-package="org.codingpedia.demo.rest

    <!-- ************ JPA configuration *********** -->
    <tx:annotation-driven transaction-manager="transactionManager"
    <bean id="transactionManager" class="org.springframework.orm.jp
        <property name="entityManagerFactory" ref="entityManagerFac
    </bean>
    <bean id="transactionManagerLegacy" class="org.springframework.
        <property name="entityManagerFactory" ref="entityManagerFac
    </bean>
    <bean id="entityManagerFactory" class="org.springframework.orm.
        <property name="persistenceXmlLocation" value="classpath:co
        <property name="persistenceUnitName" value="demoRestPersist
        <property name="dataSource" ref="restDemoDS" />
        <property name="packagesToScan" value="org.codingpedia.demo
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.Hiberna
                <property name="showSql" value="true" />
                <property name="databasePlatform" value="org.hibern
            </bean>
        </property>
    </bean>
    <bean id="entityManagerFactoryLegacy" class="org.springframewor
        <property name="persistenceXmlLocation" value="classpath:co
        <property name="persistenceUnitName" value="demoRestPersist
        <property name="dataSource" ref="restDemoLegacyDS" />
        <property name="packagesToScan" value="org.codingpedia.demo
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.Hiberna
                <property name="showSql" value="true" />
                <property name="databasePlatform" value="org.hibern
            </bean>
        </property>
    </bean>
```

```
    <bean id="podcastDao" class="org.codingpedia.demo.rest.dao.Podc
    <bean id="podcastService" class="org.codingpedia.demo.rest.serv
    <bean id="podcastsResource" class="org.codingpedia.demo.rest.re
    <bean id="podcastLegacyResource" class="org.codingpedia.demo.re

    <bean id="restDemoDS" class="org.springframework.jndi.JndiObjec
        <property name="jndiName" value="java:comp/env/jdbc/restDem
        <property name="resourceRef" value="true" />
    </bean>
    <bean id="restDemoLegacyDS" class="org.springframework.jndi.Jnd
        <property name="jndiName" value="java:comp/env/jdbc/restDem
        <property name="resourceRef" value="true" />
    </bean>
</beans>
```

其中 `podcastsResource` 是指向 REST API 实体

3. REST API (设计与实现)

3.1. 资源

3.1.1. 设计

REST 中的资源主要包括下面两大思想：

- 每个都指向了全球标示符（如，HTTP 中的 URI）
- 有一个或多个表示（我们将在本示例使用 JSON 格式）

REST 中的资源 一般是名词 (podcasts, customers, user, accounts 等) 而不是名词 (getPodcast, deleteUser 等）

本教程使用的端点有：

- `/podcasts` – （注意复数）URI标识的资源 podcasts 集合的播客
- `/podcasts/{id}` – 通过 podcasts 的ID, URI 标识一个podcasts 资源，

3.1.2. 实现

为求精简 ， podcast 只包含下列属性:

- id – podcast 的唯一标识
- feed – podcast 的 feed url
- title – 标题
- linkOnPodcastpedia – 链接
- description – 描述

我用了两种 Java 类来表示 podcast 代码，是为了避免 类及其属性/方法 被 JPA 和 XML/JAXB/JSON 的注释堆满了：

- PodcastEntity.java – JPA 注解类用在 DB 和业务层
- Podcast.java – JAXB/JSON 注解类用在外观和业务层

Podcast.java

```
package org.codingpedia.demo.rest.resource;

//imports omitted for brevity

/**
 * Podcast resource placeholder for json/xml representation
 *
 * @author ama
 *
 */
@SuppressWarnings("restriction")
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Podcast implements Serializable {

    private static final long serialVersionUID = -80396866960763370

    /** id of the podcast */
    @XmlElement(name = "id")
    private Long id;

    /** title of the podcast */
    @XmlElement(name = "title")
    private String title;

    /** link of the podcast on Podcastpedia.org */
    @XmlElement(name = "linkOnPodcastpedia")
    private String linkOnPodcastpedia;

    /** url of the feed */
    @XmlElement(name = "feed")
    private String feed;

    /** description of the podcast */
    @XmlElement(name = "description")
    private String description;

    /** insertion date in the database */
    @XmlElement(name = "insertionDate")
    @XmlJavaTypeAdapter(DateISO8601Adapter.class)
    @PodcastDetailedView
    private Date insertionDate;

    public Podcast(PodcastEntity podcastEntity){
```

```
        try {
            BeanUtils.copyProperties(this, podcastEntity);
        } catch (IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    public Podcast(String title, String linkOnPodcastpedia, String
            String description) {

        this.title = title;
        this.linkOnPodcastpedia = linkOnPodcastpedia;
        this.feed = feed;
        this.description = description;

    }

    public Podcast(){}

//getters and setters now shown for brevity
}
```

转化成 JSON 输出如下

```
{
    "id":1,
    "title":"Quarks & Co - zum Mitnehmen-modified",
    "linkOnPodcastpedia":"http://www.podcastpedia.org/podcasts/1/Qu
    "feed":"http://podcast.wdr.de/quarks.xml",
    "description":"Quarks & Co: Das Wissenschaftsmagazin",
    "insertionDate":"2014-05-30T10:26:12.00+0200"
}
```

3.2. 方法

简单的说明

- 创建 = POST
- 读 = GET
- 更新 = PUT
- 删除 = DELETE

但不是一一映射，因为 PUT 也可以创建， POST也可以用在更新。

注：读取和删除它是很清楚的，他们确实是和 *GET* 、*DELETE*一对一的映射。无论如何，*REST* 是一种架构风格，不是一个规范，你应该适应你的架构需要，但如果你想让你的 *API* 让更多的公众愿意使用它，你应该遵循一定的“最佳实践”。

`PodcastRestResource` 类 是处理所有的请求

```
package org.codingpedia.demo.rest.resource;
//imports
......................
@Component
@Path("/podcasts")
public class PodcastResource {
    @Autowired
    private PodcastService podcastService;
    .....................
}
```

注意 类定义前面的 `@Path("/podcasts")` ，所有与 podcast 关联的资源都会出现在 这个路径下。 `@Path` 注解值是关联 URI 的路径。

在上面的例子中，该 Java 类将托管在 `/podcasts` URI 路径。 `PodcastService` 接口公开的业务逻辑 到 REST 外观层。

3.2.1. 创建 podcast

3.2.1.1. 设计

常见的的方式利用 POST 创建资源，如前所述，创建一个新的资源，可以用 POST 和 PUT 的方法，我是这样做的：

| Description | URI | HTTP method | HTTP Status response | | --- | --- | --- | --- | | 增加新的 podcast | /podcasts/ | POST | 201 Created | | 增加新的 podcast (必须传所有的值) | /podcasts/{id} | PUT | 201 Created |

PUT POST 最大的区别是 ，PUT 就是把你应该事先知道资源将被创建的位置和发送所有可能值的实体。

3.2.1.2. 实现

3.2.1.2.1. POST 创建一个单资源

```
/**
 * Adds a new resource (podcast) from the given json format (at lea
 * and feed elements are required at the DB level)
 *
 * @param podcast
 * @return
 * @throws AppException
 */
@POST
@Consumes({ MediaType.APPLICATION_JSON })
@Produces({ MediaType.TEXT_HTML })
public Response createPodcast(Podcast podcast) throws AppException
    Long createPodcastId = podcastService.createPodcast(podcast);
    return Response.status(Response.Status.CREATED)// 201
            .entity("A new podcast has been created")
            .header("Location",
                    "http://localhost:8888/demo-rest-jersey-spring/
                            + String.valueOf(createPodcastId)).buil
}
```

注解

- `@POST` – 指示方法响应到 HTTP POST 请求
    - @Consumes({MediaType.APPLICATION_JSON}) – 定义方法可以接受的媒体类型，本例为"application/json"
    - @Produces({MediaType.TEXT_HTML}) – 定义方法产生的媒体类型本例为 "text/html"

响应

- 成功: HTTP 状态 为 201 的 text/html 文件和头的位置指定的资源已被创建
- 错误:
    - 400：没有足够的数据提供
    - 409：冲突了。如果在服务器端被确定 具有相同的 podcast 的存在

3.2.1.2.2. 通过 PUT 创建单资源 (“podcast”)

这将执行 更新 Podcast 处理。

3.2.1.2.3. 附加 – 通过表单创建 (“podcast”)资源

```
/**
 * Adds a new podcast (resource) from "form" (at least title and fe
 * elements are required at the DB level)
 *
 * @param title
 * @param linkOnPodcastpedia
 * @param feed
 * @param description
 * @return
 * @throws AppException
 */
@POST
@Consumes({ MediaType.APPLICATION_FORM_URLENCODED })
@Produces({ MediaType.TEXT_HTML })
@Transactional
public Response createPodcastFromApplicationFormURLencoded(
        @FormParam("title") String title,
        @FormParam("linkOnPodcastpedia") String linkOnPodcastpedia,
        @FormParam("feed") String feed,
        @FormParam("description") String description) throws AppExc

    Podcast podcast = new Podcast(title, linkOnPodcastpedia, feed,
            description);
    Long createPodcastid = podcastService.createPodcast(podcast);

    return Response
            .status(Response.Status.CREATED)// 201
            .entity("A new podcast/resource has been created at /de
                    + createPodcastid)
            .header("Location",
                    "http://localhost:8888/demo-rest-jersey-spring/
                            + String.valueOf(createPodcastid)).buil
}
```

注解

- `@POST` – 指示方法响应到 HTTP POST 请求
- `@Consumes({MediaType.APPLICATION_FORM_URLENCODED})` – 定义方法可以接受的媒体类型，本例为"application/x-www-form-urlencoded"
- `@FormParam` – 这个注解绑定的表单参数值包含了请求对应资源方法参数的实体。值是 URL 的解码，除非 禁用 解码的注解。
- `@Produces({MediaType.TEXT_HTML})` – 定义方法产生的媒体类型本例为"text/html"

响应

- 成功: HTTP 状态 为 201 的 text/html 文件和头的位置指定的资源已被创建
- 错误:
    - 400：没有足够的数据提供

- 409：冲突了。如果在服务器端被确定 具有相同的 podcast 的存在

3.2.2. 读 podcast

3.2.2.1. 设计

API 支持两种操作

- 返回 podcast 的集合
- 根据 id 返回 podcast

注意到集合资源的参数–rderByInsertionDate 和 numberDaysToLookBack。在URI查询参数添加过滤器而不是路径的一部分这个是很有道理的。

3.2.2.2. 实现

3.2.2.2.1. 获取所有 podcasts (“/”)

```
/**
 * Returns all resources (podcasts) from the database
 *
 * @return
 * @throws IOException
 * @throws JsonMappingException
 * @throws JsonGenerationException
 * @throws AppException
 */
@GET
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML ]
public List<Podcast> getPodcasts(
        @QueryParam("orderByInsertionDate") String orderByInsertion
        @QueryParam("numberDaysToLookBack") Integer numberDaysToLoc
        throws JsonGenerationException, JsonMappingException, IOExc
        AppException {
    List<Podcast> podcasts = podcastService.getPodcasts(
            orderByInsertionDate, numberDaysToLookBack);
    return podcasts;
}
```

注解

- `@GET` – 指示方法响应到 HTTP GET 请求
- `@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML` – 定义方法可以接受的媒体类型，本例为"application/json" 或者 "application/xml"（在 Podcast 类前 添加 `@XmlRootElement` ），将返回 JSON 或者 XML 格式的 podcast 集合

响应

- 成功: HTTP 状态 为 200 的 podcast 数据集合

3.2.2.2.1. 读一个 podcast

根据 id 获取一个 podcast

```
@GET
@Path("{id}")
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML ]
public Response getPodcastById(@PathParam("id") Long id)
        throws JsonGenerationException, JsonMappingException, IOExc
        AppException {
    Podcast podcastById = podcastService.getPodcastById(id);
    return Response.status(200).entity(podcastById)
            .header("Access-Control-Allow-Headers", "X-extra-header
            .allow("OPTIONS").build();
}
```

注解

- `@GET` – 指示方法响应到 HTTP GET 请求
- `@PathParam("id")` - 绑定传递的参数值
- `@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML` – 定义方法可以接受的媒体类型，本例为"application/json" 或者 "application/xml"（在 Podcast 类前 添加 `@XmlRootElement` ），将返回 JSON 或者 XML 格式的 podcast 集合

响应

- 成功: HTTP 状态 为 200 的 podcast
- 错误： 404 Not found。如果没有在数据库中找到

3.2.3. 更新 podcast

3.2.3.1. 设计

| Description | URI | HTTP method | HTTP Status response | | --- | --- | --- | --- | | 更新 podcast (完全) | /podcasts/{id} | PUT | 200 OK | | 更新 podcast (部分) | /podcasts/{id} | POST | 200 OK |

1.完全更新 – 提供所有的值 2.部分更新 – 传递部分属性值即可

3.2.3.1. 实现

3.2.3.1.1. 完全更新

创建或者完全更新资源

```
@PUT
@Path("{id}")
@Consumes({ MediaType.APPLICATION_JSON })
@Produces({ MediaType.TEXT_HTML })
public Response putPodcastById(@PathParam("id") Long id, Podcast po
        throws AppException {

    Podcast podcastById = podcastService.verifyPodcastExistenceById

    if (podcastById == null) {
        // resource not existent yet, and should be created under t
        // specified URI
        Long createPodcastId = podcastService.createPodcast(podcast
        return Response
                .status(Response.Status.CREATED)
                // 201
                .entity("A new podcast has been created AT THE LOCA
                .header("Location",
                        "http://localhost:8888/demo-rest-jersey-spr
                                + String.valueOf(createPodcastId)).
    } else {
        // resource is existent and a full update should occur
        podcastService.updateFullyPodcast(podcast);
        return Response
                .status(Response.Status.OK)
                // 200
                .entity("The podcast you specified has been fully u
                .header("Location",
                        "http://localhost:8888/demo-rest-jersey-spr
                                + String.valueOf(id)).build();
    }
}
```

注解

- `@PUT` – 指示方法响应到 HTTP PUT 请求
- `@PathParam("id")` - 绑定传递的参数值
- `@Consumes({MediaType.APPLICATION_JSON})` – 定义方法可以接受的媒体类型，本例为"application/json"
- `@Produces({MediaType.TEXT_HTML})` – 定义方法可以产生的媒体类型，本例为t"ext/html"

响应

- 创建
  - 成功: HTTP 状态 为 201 Created
  - 错误： 400 Bad Request。如果需要的属性值没有提供
- 完全更新：
  - 成功: HTTP 状态 为 200

- 错误： 400 Bad Request。如果不是所有的属性都提供

3.2.3.1.2. 部分更新

```
//PARTIAL update
@POST
@Path("{id}")
@Consumes({ MediaType.APPLICATION_JSON })
@Produces({ MediaType.TEXT_HTML })
public Response partialUpdatePodcast(@PathParam("id") Long id, Podc
    podcast.setId(id);
    podcastService.updatePartiallyPodcast(podcast);
    return Response.status(Response.Status.OK)// 200
            .entity("The podcast you specified has been successfull
            .build();
}
```

注解

- `@POST` – 指示方法响应到 HTTP POST 请求
- `@PathParam("id")` - 绑定传递的参数值
- `@Consumes({MediaType.APPLICATION_JSON})` – 定义方法可以接受的媒体类型，本例为"application/json"
- `@Produces({MediaType.TEXT_HTML})` – 定义方法可以产生的媒体类型，本例为t"ext/html"

响应

- 成功: HTTP 状态 为 200 OK
- 错误： 404 Not Found。如果资源不存在

3.2.4. 删除 podcast

3.2.4.1. 设计

| Description | URI | HTTP method | HTTP Status response | | --- | --- | --- | --- | | 移除所有 podcasts | /podcasts/ | DELETE | 204 No content | | 移除特定位置的 podcast | /podcasts/{id} | DELETE | 204 No content |

3.2.4.2. 实现

3.2.4.2.1. 删除所有资源

```
@DELETE
@Produces({ MediaType.TEXT_HTML })
public Response deletePodcasts() {
    podcastService.deletePodcasts();
    return Response.status(Response.Status.NO_CONTENT)// 204
            .entity("All podcasts have been successfully removed").
}
```

注解

- `@DELETE` – 指示方法响应到 HTTP DELETE 请求
- `@Produces({MediaType.TEXT_HTML})` – 定义方法可以产生的媒体类型，本例为"text/html"

响应

- 返回 html 文档

3.2.4.2.2. 删除一个资源

```
@DELETE
@Path("{id}")
@Produces({ MediaType.TEXT_HTML })
public Response deletePodcastById(@PathParam("id") Long id) {
    podcastService.deletePodcastById(id);
    return Response.status(Response.Status.NO_CONTENT)// 204
            .entity("Podcast successfully removed from database").b
}
```

注解

- `@DELETE` – 指示方法响应到 HTTP DELETE 请求
- `@PathParam("id")` - 绑定传递的参数值
- `@Consumes({MediaType.APPLICATION_JSON})` – 定义方法可以接受的媒体类型，本例为"application/json"
- `@Produces({MediaType.TEXT_HTML})` – 定义方法可以产生的媒体类型，本例为"text/html"

响应

- 成功: HTTP 状态 为 204 No Content
- 错误： 404 Not Found。如果资源不存在

4. 日志

详见 http://www.codingpedia.org/ama/how-to-log-in-spring-with-slf4j-and-logback/

5. 异常处理

错误处理要有统一的格式，就像下面

```
{
   "status": 400,
   "code": 400,
   "message": "Provided data not sufficient for insertion",
   "link": "http://www.codingpedia.org/ama/tutorial-rest-api-design
   "developerMessage": "Please verify that the feed is properly gen
}
```

6. 服务端添加 CORS 支持

7. 测试

7.1. 在Java集成测试

7.1.1. 配置

7.1.1.1 Jersey 客户端依赖

```
<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>${jersey.version}</version>
    <scope>test</scope>
</dependency>
```

7.1.1.2. Failsafe 插件

```
<plugins>
    [...]
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.16</version>
        <executions>
            <execution>
                <id>integration-test</id>
                <goals>
                    <goal>integration-test</goal>
                </goals>
            </execution>
            <execution>
                <id>verify</id>
                <goals>
                    <goal>verify</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
    [...]
</plugins>
```

7.1.1.2. Jetty Maven 插件

```
<plugins>
    <plugin>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>${jetty.version}</version>
        <configuration>
            <jettyConfig>${project.basedir}/src/main/resources/conf
            <stopKey>STOP</stopKey>
            <stopPort>9999</stopPort>
            <stopWait>5</stopWait>
            <scanIntervalSeconds>5</scanIntervalSeconds>
        [...]
        </configuration>
        <executions>
            <execution>
                <id>start-jetty</id>
                <phase>pre-integration-test</phase>
                <goals>
                    <!-- stop any previous instance to free up the
                    <goal>stop</goal>
                    <goal>run-exploded</goal>
                </goals>
                <configuration>
                    <scanIntervalSeconds>0</scanIntervalSeconds>
                    <daemon>true</daemon>
                </configuration>
            </execution>
            <execution>
                <id>stop-jetty</id>
                <phase>post-integration-test</phase>
                <goals>
                    <goal>stop</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
    [...]
</plugins>
```

详细配置见源码中的 pom.xml

7.1.2. 编译集成测试

使用 JUnit 作为测试框架。默认的 Failsafe 插件 自动包含所有测试类

- " `_*/IT_.java` " – “IT”开头的文件.
- " `_*/_IT.java` " – “IT”结尾的文件.
- " `_*/_ITCase.java` " – “ITCase”结尾的文件.

创建了测试类 RestDemoServiceIT

```
public class RestDemoServiceIT {

    [....]
    @Test
    public void testGetPodcast() throws JsonGenerationException,
            JsonMappingException, IOException {

        ClientConfig clientConfig = new ClientConfig();
        clientConfig.register(JacksonFeature.class);

        Client client = ClientBuilder.newClient(clientConfig);

        WebTarget webTarget = client
                .target("http://localhost:8888/demo-rest-jersey-spr

        Builder request = webTarget.request(MediaType.APPLICATION_J

        Response response = request.get();
        Assert.assertTrue(response.getStatus() == 200);

        Podcast podcast = response.readEntity(Podcast.class);

        ObjectMapper mapper = new ObjectMapper();
        System.out
                .print("Received podcast from database ***********
                        + mapper.writerWithDefaultPrettyPrinter()
                                .writeValueAsString(podcast));

    }
}
```

注意：

- 在客户也要注册 `JacksonFeature` ，这样才能解析 JSON格式
- 用 jetty 测试，端口 8888
- 期望 返回 200 状态 给我们的请求
- `org.codehaus.jackson.map.ObjectMapper` 帮助返回格式化的 JSON

7.1.3. 运行集成测试

运行

```
mvn verify
```

设置 `jetty.port` 属性到 8888,Eclipse 配置如下

7.2. 用 SoapUI 集成测试

youtube视频教程（需翻墙）

8. 版本管理

几个要点：

- `URL: “/v1/podcasts/{id}”`
- `Accept/Content-type header: application/json; version=1`

在 路径中 加入 版本信息

```
@Component
@Path("/v1/podcasts")
public class PodcastResource {...}
```

参考：

- https://jersey.java.net/
- https://github.com/waylau/Jersey-2.x-User-Guide
- http://www.codingpedia.org/ama/tutorial-rest-api-design-and-implementation-in-java-with-jersey-and-spring/